Applications and Implementations of Static Discontinuity Grammars

Veronica Dahl and
Fred Popowich

CSS/LCCR TR 89-10

92-05570/

Applications and Implementations of Static Discontinuity Grammars

Veronica Dahl
School of Computing Science
Centre for Systems Science
Laboratory for Computer and Communications Research
Simon Fraser University
Burnaby, British Columbia
V5A 1S6 Canada
veronica@cs.sfu.ca (604) 291-3372

Fred Popowich
School of Computing Science
Centre for Systems Science
Simon Fraser University
Burnaby, British Columbia
V5A 1S6 Canada
Fred_Popowich@SFU.BITNET (604) 291-4193

Abstract

Static Discontinuity Grammar (SDG) is a type of logic grammar. Its distinctive features are that rules can state dependencies between any two or more subtrees in a derivation tree for a grammatical sentence, and that descriptions are expressed in terms of node domination in the tree. These features make SDGs useful in formulations of constraints utilized in linguistic theories such as Government and Binding. In this paper, we will look at how linguistic and non-linguistic constraints can be captured succinctly in SDGs. Simple examples will illustrate linguistic ways of thinking without requiring detailed knowledge of any particular linguistic theory, and will also illustrate some non-linguistic possibilities of SDGs. We will then look at different ways of processing the grammars introduced in these examples. In particular, implementations of top-down depth-first, and left corner SDG interpreters will be examined, as well as a compiler that generates code for both analysis and synthesis.

Key Words: parsing, generation, logic grammars, discontinuous constituents, linguistic constraints.

This research is supported by Fred Popowich's research fellowship from the Advanced Systems Institute of British Columbia and by his Natural Sciences and Engineering Research Council of Canada Grant No. OGP0041910, by Veronica Dahl's NSERC grant 06-4244, and also partially by her Center for Systems Science Grant No. 4038, in view of collaboration with Centre Canadien de Recherche en Informatisation du Travail. The views expressed in this article are strictly those of the authors.

1. Introduction

The proliferation of different grammar formalisms used in computational linguistics is exceeded only by the number of implementations for these various formalisms. However, underlying all these different formalisms are common approaches which are merely specialized for the particular domain of interest or extended to allow certain linguistic phenomena to be described more easily. Similarly, the various implementations tend to rely on sound general parsing techniques which are refined for the particular formalism.

Static Discontinuity Grammars (SDGs) belong to a family of grammars called logic grammars and closely resemble what one might call a logic grammar rendition of Scattered Context Grammars (Greibach and Hopcroft, 1969). The first logic grammar was introduced by Colmerauer (1978) and became widely known through Pereira and Warren's (1980) restricted version, Definite Clause Grammars. SDGs evolved from Dahl's Discontinuous Grammars (Dahl, 1984, Dahl and Abramson, 1984, Popowich, 1986, Dahl and Saint-Dizier, 1986, Dahl, 1988a)[1] which are a generalization of Pereira's (1981) Extraposition Grammars.

In Discontinuous Grammars (DGs), unidentified strings of symbols, called skips, can be arbitrarily repositioned (or duplicated, or deleted) by a rule. Its rules can therefore flexibly relate specific non-contiguous constituents (for purposes such as movement, long-distance dependency, coindexing) while skipping over, without analyzing, any discontinuities found in between them (Dahl and Abramson, 1984, Dahl, 1984, Dahl, 1988a).

SDGs are a subset of DGs, devised by Dahl (1986), which can skip over intermediate strings of symbols without moving them (Dahl, 1986, Dahl, 1988b, Dahl, 1988c), and which can dynamically enforce constraints described statically in terms of primitives such as node domination in a hierarchy. Such straightforward treatment of constraints follows from the static discontinuity feature, which allows for a sentence's history of analysis or synthesis to be shaped as a tree rather than a graph, thus accommodating the typically hierarchically expressed constraints of contemporary linguistic theories. This feature also allows us easier and more efficient implementations, as well as greater perspicuity, than DGs.

This article reviews the background, presents a view of SDGs in terms of admissibility conditions on derivation trees, motivates SDGs through two simple applications, and discusses three implementations: a top-down interpreter, a left-corner parser with techniques for avoiding spurious ambiguities and for improving efficiency, and a compiler into reversible grammar processors (i.e., usable either for analysis or generation). While our discussion focusses on grammars, it applies equally to logic *programming* with static discontinuities (cf. section 7).

---

[1]Discontinuous Grammars were formerly called Gapping Grammars.

2. Background

The first implementation for the general DG formalism was a compiler called SYNAL, coded by McCord from a design by Dahl and tested jointly, in view of its use for the later discarded purpose of treating coordination in natural language (Dahl and McCord, 1983). SYNAL is very concise but inefficient, since it determines what strings to skip by sequentially trying strings of length 0, 1, 2, etc., and backtracking upon failure. Its complete description can be found in (Dahl and Abramson, 1984).

A more efficient but less encompassing SYNAL version was proposed by Abramson and also studied in (Dahl and Abramson, 1984). It limits the formalism to rules with only one skip string, which can only be followed by a single terminal, and which must rewrite at the rightmost position of the right hand side. In other words, rules have the form shown in (1), with α and β strings in V^*, and $nt \in V_N$, with V_N corresponding to the set of nonterminal symbols of a grammar and V to the set of all terminals and nonterminals.

(1) nt, α, skip(X), [terminal] $\rightarrow$ β, skip(X).

This is very restrictive, but still more flexible than Metamorphosis Grammars (Colmerauer, 1978), and the implementation is an interesting application of message passing.

A third implementation was developed by Popowich (1986). This is a shift-reduce parser which operates in a bottom-up manner and can therefore afford to allow terminals as well as nonterminals to be the leading symbol of a rule.

A further SYNAL version was developed jointly by Dahl and Saint-Dizier (1986), to handle user-defined constraints expressed as the interdiction to move any node outside a specified domain in the parse history. Although this first attempt did allow some linguistic constraints to be expressed and thus reduce the consideration of ungrammatical sentences, it had the following drawbacks:

- the system worked only for analysis, and not for generation,
- as in all previous logic grammars admitting more than one left-hand side symbol, parse graphs rather than trees were produced, thus requiring an inelegant artifice to keep track of hierarchical relationships, often crucial to linguistic constraints in general,
- a further artifice was needed to avoid loops,
- skipped substrings were tried in the inefficient order of increasing length causing, in general, much backtracking, and
- the constraint format was not specifically tailored for the typical constraints of Government-Binding theory (Chomsky, 1982).

In order to solve these problems, and in view of Dahl's research on Government-Binding theory and on its use for generating English error messages from conceptual graph

representations, she then developed the more efficiently implementable family of Static Discontinuity Grammars.

3. The tree admissibility view of SDGs

An SDG possesses two types of grammar rules. *Type-one* rules are essentially Definite Clause Grammar rules. That is, they are of the form

(2) $\alpha \rightarrow \beta_1, \beta_2, \ldots, \beta_n$

for $n \geq 0$ where α and β_i are *terms*. The case where $n=0$ is represented by the appearance of [ ] on the right hand side of the rule. Terminals are noted between square brackets. A term consists of a predicate symbol followed by zero or more arguments. Each argument may either be a constant, a variable, or a term. Following the convention adopted in logic grammars, constants and predicate symbols will be denoted by lower case words, while variables will be represented by upper case words. Like many other logic grammar formalisms, arbitrary procedure calls, enclosed in braces, may also appear on the right hand side of the rule. An example of a type-one rule is shown in (3).

(3) s → np(Number_np), vp(Number_vp), { agree(Number_np, Number_vp) }

Type-two rules consist of a sequence of rules of the first type (enclosed in braces). That is, the rules have the following form.

(4) $\{ \alpha_1 \rightarrow \beta_{1_1}, \beta_{1_2}, \ldots, \beta_{1_n}$
$\alpha_2 \rightarrow \beta_{2_1}, \beta_{2_2}, \ldots, \beta_{2_n}$
$\vdots$
$\alpha_m \rightarrow \beta_{m_1}, \beta_{m_2}, \ldots, \beta_{m_n} \}$

Rules of this type can be applied in parallel to all the symbols α_i that appear in the correct order somewhere in a derivation. So the above rule can be interpreted as shown in (5) where each symbol γ_i corresponds to a sequence of zero or more terminal or nonterminal symbols.

(5) $\alpha_1, \gamma_1, \alpha_2, \gamma_2, \ldots \alpha_m \rightarrow \beta_{1_1}, \beta_{1_2}, \ldots, \beta_{1_n}, \gamma_1, \beta_{2_1}, \beta_{2_2}, \ldots, \beta_{2_n}, \gamma_2, \ldots \beta_{m_1}, \beta_{m_2}, \ldots, \beta_{m_n}$

This is an example in terms of the rule format used in Discontinuous Grammars. Because Discontinuous Grammar rules can delete, move or copy the skipped regions γ_i into arbitrary places, these regions must be explicitly named in the general Discontinuous Grammar formalism, but they may be implicit in SDGs.

Rules like (5) can be viewed as rewrite or production rules; the semantics of an SDG rule can be defined in terms of a rewriting relation '$\Rightarrow$' on a sequence of grammar symbols. Formal accounts of this view have been given in (Dahl, 1989). Roughly speaking, for any two sequences of grammar symbols *u* and *v*, $u \Rightarrow v$ iff there exists a production rule like (5) and a substitution θ of values for variables such that:

i. $u\theta = (\alpha_1, \gamma_1, \alpha_2, \gamma_2, \ldots \alpha_m)\theta$

ii. $v\theta = (\beta_{1_1}, \beta_{1_2}, \ldots, \beta_{1_n}, \gamma_1, \beta_{2_1}, \beta_{2_2}, \ldots, \beta_{2_n}, \gamma_2, \ldots \beta_{m_1}, \beta_{m_2}, \ldots, \beta_{m_n})\theta$

Alternatively, SDG rules can be viewed as tree admissibility conditions. This view of grammar rules is adopted in, for example, Generalized Phrase Structure Grammar (Gazdar et.al., 1985). Some existing SDG implementations also adopt this view as we shall see. Before providing a formal specification of the tree admissibility interpretation of SDG grammar rules, let us look at this interpretation in informal terms.

A type-one rule like (2) acts as an admissibility condition for a local subtree (ie., a subtree of depth one consisting of just a mother node and its daughters) of the form shown in Figure 1. To

Figure 1: Local Tree

simplify the tree nodes, we have adopted the notation α' as an abbreviation for $\alpha\theta$. Each nonterminal node in a derivation tree needs to be licensed by the invocation of a grammar rule, and the invocation of the grammar rule requires the local tree rooted by that nonterminal node to be admitted by the grammar rule. The terminal nodes of the derivation tree license (or are licensed by) a sentence from the language.

A type-two rule like (4) introduces admissibility conditions on independent local trees. Two local trees are independent in a derivation tree iff no node from one local tree dominates a node from the other local tree in the derivation tree. Equivalently, two local trees are independent iff there is a *cut* of the derivation tree which contains the root nodes of both local trees. A *cut* roughly corresponds to a path drawn left to right through a tree passing only through tree nodes and not tree edges (Aho and Ullman, 1972). These definitions can be extended for sets of any number of local trees. So for a type-two rule of the form introduced in (4), we have a set of trees as shown in Figure 2.

By treating SDG rules as admissibility conditions instead of rewrite rules, we can obtain derivational structures which would not be possible under the rewrite rule interpretation. An

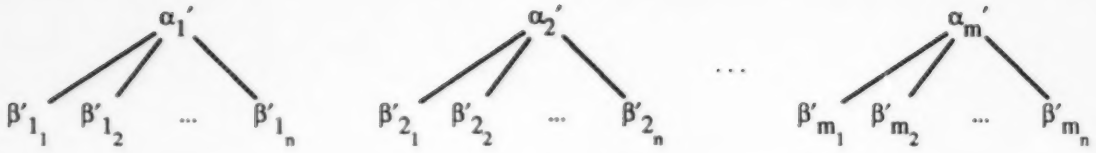

Figure 2: Sets of Independent Local Trees

SDG for the language $a_i^n b_i^n$ where each a_i is co-indexed with one and only one b_i[2] is introduced in (6).

(6) s → as, bs

{ as → [ ]
bs → [ ] }

{ as → [a(X)], as
bs → [b(X)], bs }

Under the rewrite interpretation of an SDG, the grammar in (6) describes the language where the a_i and the b_i must be in the same order. Under the admissibility interpretation, any pair of *a*'s and *b*'s may be coindexed. Thus the sentence $a_1a_2b_2b_1$ is also allowed as illustrated in Figure 3. One application of the third grammar rule admits the two independent local subtrees darkly circled in Figure 3, while another invocation of this rule admits the independent local subtrees lightly circled.

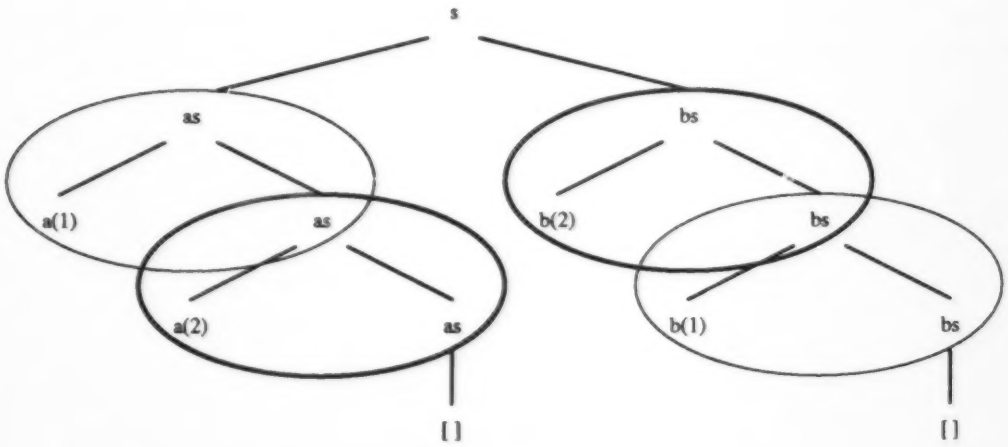

Figure 3: Derivation Tree

[2]In our grammar, the terminal symbol a_X for some *X* will be represented as *a(X)*.

Under the tree admissibility interpretation, if we only wanted to get the string where the *a*'s and *b*'s were in the same order, we could use the grammar in (7) instead.

(7) s → as([ ]), bs([ ]).

{ as(D) → [ ]
  bs(D) → [ ] }

{ as(D) → [a(X)], as([d|D])
  bs(D) → [b(X)], bs([d|D]) }

In this grammar, the nonterminal symbols possess an argument for encoding their depth in the derivation tree; the length of the list appears as the argument of a nonterminal corresponds to its depth in the tree. Each type-two grammar rule requires the root symbols of each of its associated local trees to be from the same depth *D* in the tree.

4. Definitions

We will now provide a more formal definition of the language associated with an SDG in terms of derivation trees and local trees. Although it would be possible to avoid the use of local trees in this formal definition, the use of these contructs will allow our formal definition to parallel our informal discussion and will make the definition easier to follow.

> **Definition 1:** A sentence is part of the language of a grammar iff (i) there is a derivation tree Δ which has the sentence as its fringe and (ii) there is a multiset[3] T of local trees which covers the derivation tree.

Let us once again consider the grammar introduced in (6), and the derivation tree (Figure 3) which has the sentence $a_1a_2b_2b_1$ as its fringe. The multiset of local trees covering this derivation tree is shown in Figure 4. Note that in this multiset, no tree occurs more than once. If

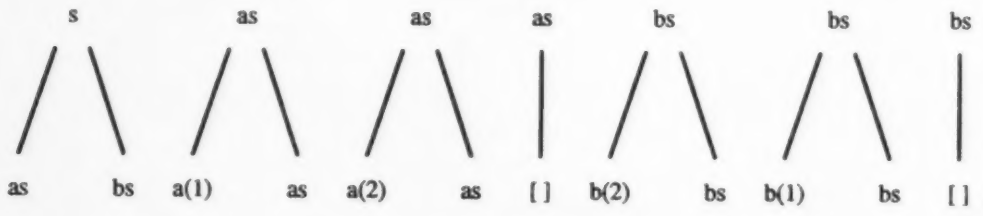

Figure 4: Derivation Tree Cover

our grammar were as shown in (8), then the cover of the derivation tree for the sentence "a a b b" would be as shown in Figure 5; some local trees would appear more than once.

---

[3]Multisets are sets in which an element can occur more than once. Their properties are discussed in (Knuth, 1969). To distinguish multisets from traditional sets in our notation, will be introduce an *m* subscript on multiset braces. The multiset union will be represented by the use of the symbol ⊎.

(8) s → as, bs

{ as → [ ]
  bs → [ ] }

{ as → [a], as
  bs → [b], bs }

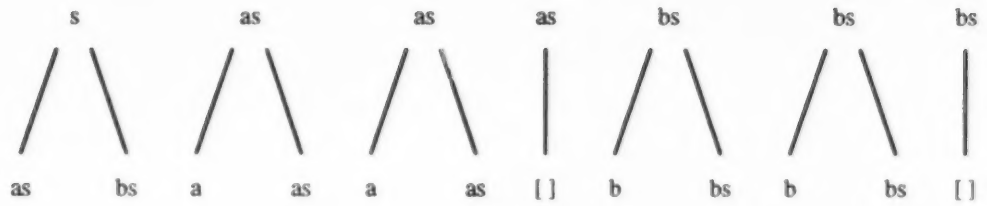

Figure 5: Multiset containing Duplicate Local Trees

Now, what exactly does it mean for a multiset of local trees to cover a derivation tree. For any local tree $\tau \in T$, let $r(\tau)$ be the root node of τ, and for any nonterminal node n in a tree let $d(n)$ be an ordered sequence corresponding to the daughter nodes of n.

Definition 2: A multiset T of local trees covers a derivation tree Δ iff there is a one to one mapping c from members of T to the nonterminal nodes of Δ that satisfies the following conditions.

- $r(\tau) = c(\tau)$; the root node of each local tree corresponds to one and only one node in the derivation tree;
- $d(r(\tau)) = d(c(\tau))$; the daughters of $r(\tau)$ and $c(\tau)$ are the same.

Such a mapping from the multiset introduced in Figure 4 to the nonterminal nodes of Figure 3 can be achieved by traversing the derivation tree in a top-down depth-first manner and pairing the nonterminal nodes with the trees as ordered in Figure 4.

For any grammar rule ρ and any substitution function θ, let t be a function from pairs of grammar rules and substitution functions to multisets of local trees; that is $t:<\rho,\theta> \rightarrow T$. For a type-one grammar rule ρ of the form shown in (2), $t(<\rho,\theta>) = \{\tau\}_m$ where $r(\tau)=\alpha\theta$ and $d(r(\tau)) = <\beta_1\theta, \beta_2\theta, ..., \beta_n\theta>$. For a type-two grammar rule ρ of the form shown in (4), $t(<\rho,\theta>) = \{\tau_1,\tau_2,...,\tau_m\}_m$ where $r(\tau_j)=\alpha_j\theta$ and $d(r(\tau_j))=<\beta_{i_1}\theta, \beta_{i_2}\theta, ..., \beta_{i_n}\theta>$. As an example, consider the three grammar rules of (6) which shall be referenced as ρ_1, ρ_2 and ρ_3. If the function $\theta_{X=1}$ assigns the value *1* to the variable *X*, then the value of $t(<\rho_3,\theta_{X=1}>)$ will correspond to the multiset shown in Figure 6. Note that we need a multiset in our definition since it is possible for a type-two rule to introduce two identical local trees.

In order to capture some sort of ordering relationship in the grammar, we shall introduce the relation <<. For any two nodes α and β in a derivation tree, $\alpha << \beta$ iff (i) α does not dominate β; (ii) β does not dominate α; (iii) there exists some node γ that dominates both α and β; and (iv)

Figure 6: Multiset associated with a Grammar Rule

there exists nodes δ_1 and δ_2 which are daughters of γ such that δ_1 precedes δ_2, δ_1 dominates α and δ_2 dominates β. *Dominates* is a reflexive transitive relationship that holds between a node and its descendents. Intuitively, the relation << states that the first node is to the left of the second node in a *cut* of a derivation tree.

Definition 3: A grammar admits a multiset of local trees T for a derivation tree Δ iff

- there exists a multiset Π of pairs of grammar rules and substitution functions $\Pi = \{<\rho_1,\theta_1>, <\rho_2,\theta_2>, \ldots <\rho_m,\theta_m>\}_m$ such that $T = \uplus_i \mathcal{A}(<\rho_i,\theta_i>)$ where $<\rho_i,\theta_i> \in \Pi$;
- for any α_j and α_k from a type-two grammar rule where $j < k$, $\mathcal{C}(\mathcal{A}(<\alpha_j,\theta_j>)) << \mathcal{C}(\mathcal{A}(<\alpha_k,\theta_j>))$ in Δ (ie. if α_j precedes α_k in a grammar rule, then the node corresponding to α_j in Δ must be to the left of the node corresponding to α_k).

So to obtain the multiset of trees that covers the derivation tree in Figure 3, we would have the following multiset of grammar rule and substitution function pairs.

(9) $\Pi = \{ <\rho_1,\theta_0>, <\rho_2,\theta_0>, <\rho_3,\theta_{X=1}>, <\rho_3,\theta_{X=2}> \}_m$

The first two rules of (6) are used once each, and since they do not contain any variables no values need to be assigned by the substitution function (as represented by θ_0). The last rule is used once with the substitution that X=1 and once with X=2.

Conceivably, we may want to relax certain clauses of the above definition to allow a grammar to describe a larger set of sentences. As we shall see later, some implementations incorporate a more general form of the << relation within their parsing strategy.

Treating the rules as well-formedness conditions on trees allows us to propose some natural extensions to the SDG to include other conditions on trees as well. We can include a set of *dominates* conditions which would be required to hold between the different nodes in a wellformed derivation tree. Such conditions would be useful for capturing certain linguistic constraints, as explained in section 5.3.

Although the tree admissibility interpretation of grammar rules may seem difficult to incorporate into a parser, it is in fact easier to handle than the case where the rules are treated as rewrite relations. As it turns out, the prototype parser discussed in (Dahl and Massicotte,

1988) treats grammars in this tree admissibility manner. All of the processors which we will be discussing work for languages described by the admissibility rules.

5. Some Applications of SDGs

The nature of type-two SDG rules makes them extremely useful for capturing in a single grammar rule relationships that require numerous rules in other grammar formalisms. In this section, we will examine a non-linguistic application of SDGs and look at some simple linguistic applications which do not require any detailed knowledge of a particular linguistic theory.

5.1. A Non-Linguistic Application

In (Dahl, 1984) the following Discontinuous Grammar is presented, describing all permutations of *n* occurrences of three symbols *a*, *b* and *c* (the "scrambled $a^n\ b^n\ c^n$ " problem):

(10) s → as, bs, cs, s.
s → [ ].

as, skip(X) → skip(X), [a].
bs, skip(X) → skip(X), [b].
cs, skip(X) → skip(X), [c].

In this grammar, the skipped regions are moved and therefore need to be represented explicitly (this is the main difference between the general Discontinuous Grammar formalism and SDGs). Intuitively, a given expected symbol (say, *a*) may not appear at the point in which its corresponding nonterminal symbol (say, *as*) expects it, but may have to be found after a series of skipped symbols, represented by *X*. The skipped symbols remain available for future analysis.

If we are permuting *m* symbols, this formulation needs *m*+2 rules, as opposed to the *m*!+1 rules needed by a simple DCG formulation[4]. But thinking in terms of discontinuities that are *static* allows us not only a simpler formulation (in the sense that it is not cluttered with explicit *skip* symbols), but also a shorter one, in which the number of rules needed (four) is constant regardless of *m* as shown in the SDG in (11).

[4]e.g. for m=3, s → []; s → [a,b,c],s; s → [a,c,b],s; s → [b,a,c],s; s → [b,c,a],s; s → [c,a,b],s; s → [c,b,a],s.

(11) s → movable(a), movable(b), movable(c), s

s → [ ]

movable(X) → [X]

{ movable(X1) → [X2], {diff(X1,X2)}
movable(X2) → [X1] }

The type-two rule accepts an *X2* symbol which is different from an expected *X1* symbol (as reflected by the presence of the restriction *diff(X1,X2)*), but changes a future expectation of *X2* into an expectation of *X1*. Adding more symbols to permute only affects the first rule for *s*.

5.2. A Linguistic Application

The general technique described above for permutations can be adapted to the linguistic problem of free word order, namely, how to represent concisely the admission, by some natural languages, of different possible orderings of words in a sentence.

We next show an SDG grammar for a contrived example in Latin. Consider the following sentence.

(12) Puella bona puerum parvum amat.
Good girl small boy loves
'The good girl loves the small boy'

The noun and adjective in the subject and/or object noun phrase may be discontinued as shown in (13).

(13) Puella puerum amat bona parvum.

A simple SDG grammar which captures all the different possibilities for word order variation is provided in (14) along with the 'dictionary entries' of the various words. Each entry contains information about the word's syntactic category and case as required by the grammar rules.

(14) sentence → np(nom), np(acc), mov(verb,[ ])

np(Case) → mov(adj,Case), mov(noun,Case)

mov(Category, Case) → [Word], {dict(Category,Case,Word)}

{ mov(Cat,Case) → [Word1], {dict(Cat1,Case1,Word1)}
mov(Cat1,Case1) → [Word], {dict(Cat,Case,Word)} }

```
dict(verb,[],amat).
dict(noun,acc,puerum).
dict(noun,nom,puella).
dict(adj,acc,parvum).
dict(adj,nom,bona).
```

In the Discontinuous Grammar formulation presented in (Dahl, 1984), the number of rules needed grows linearly depending on the number of constituents which can move freely. Here, as in the permutation example, two rules handle all movable constituents, which appear as arguments of the *mov* predicate. In the case of the *verb* constituent, the second argument of *mov* is only kept for uniformity, since verbs have no case.

5.3. SDG Constraints

Let us now imagine a linguistic constraint that prohibits the movement of a constituent into a position outside of the innermost sentence node that dominates it. For instance, if our sample sentence above were imbricated in another one, its derivation tree would look as shown in Figure 7 (*mov* is abbreviated as *m*).

The movable constituents shown are interchangeable with each other, but must remain within the innermost sentence. This constraint can be easily expressed in SDGs, as shown in (15), by defining a predicate *block(X,P)* (block the movement of a constituent *X* into a position *P*), in terms of a primitive predicate *dominates(n1,n2)* (node *n1* dominates node *n2* in the derivation tree):

```
(15) block(X,P) :-  dominates(sent(S1), sent(S2)),
                    dominates(sent(S2), X),
                    not(dominates(sent(S2),P))).
```

Complete SDG grammar development systems would include a precompiler to translate such constraints into conditions on type-two rule application, as well as an automatic derivation-tree constructing and consulting mechanism that would be used to dynamically enforce these statically defined constraints (a type-two rule's application is allowed if it does not attempt a movement forbidden by a constraint).

The type-two rule in (14), for instance, would incorporate the constraint as soon as both

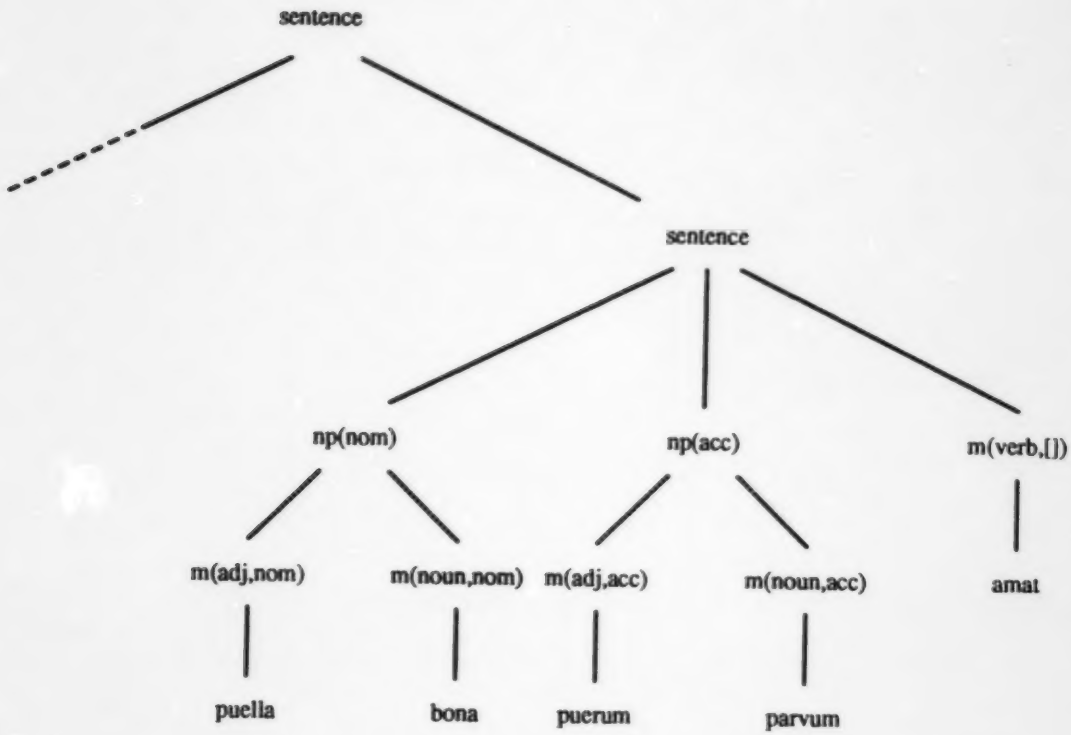

Figure 7: Derivation Tree for Latin Sentence

categories involved in movement are known:[5]

(16) { mov(Cat,Case) → [Word1], {dict(Cat1,Case1,Word1),not(block(Cat,Cat1))}
mov(Cat1,Case1) → [Word], {dict(Cat,Case,Word)} }

Of course, neither the language nor the constraint presented as examples are very realistic from a linguistic point of view. However, both the problem of free word order and the general format of our sample constraint are encountered in linguistic reality. Hopefully, we have been able to convey some main points in SDG programming for contemporary linguistic theory without getting into the complexities of any theory in particular. Further details concerning the specific subject of implementing constraints can be found in (Dahl and Massicotte, 1988).

[5]Tree construction arguments are not shown, being quite straightforward but cumbersome to read.

6. Implementations

In introducing some SDG grammars in the previous sections we have not assumed any one particular processing strategy (nor should we have). There are a variety of different ways to use these grammars in natural language processors. In this section, we shall consider how an SDG specification can be used by a top-down parser and by a left-corner parser. We shall then see how such a specification can be compiled into a Prolog program that can be used both as a parser and as a generator.

6.1. Top-Down Parsing

In (Dahl and Massicotte, 1988), a short, simple top-down depth-first backtracking interpreter is introduced, which corresponds to the tree admissibility interpretation of SDGs. The implementation of dynamically enforced linguistic constraints is also discussed in (Dahl and Massicotte, 1988), but in this article we shall disregard this aspect. A slightly modified version of this interpreter is introduced in Figure 8.

```
% sdparse(List of Words, List of Goals, List of Pending Rules)

sdparse([],[],[]) :- !.
sdparse([T1|T],[T1|S],P):- sdparse(T,S,P).     % remove a terminal symbol

sdparse(S,[H|G],P):- getsubrule(P,[H|B],P1),   % pending rule for current goal
                     concat(B,G,G1),
                     sdparse(S,G1,P1).

sdparse(S,[H|G],P):- sdrule(H,B,Sr),           % apply a type 2 rule
                     concat(B,G,G1),           % add subrules to pending list
                     concat(Sr,P,P1),
                     sdparse(S,G1,P1).

sdparse(S,[H|G],P):- sdrule(H,B),              % apply a type 1 rule
                     concat(B,G,G1),
                     sdparse(S,G1,P).

% getsubrule(List of pending rules, subrule chosen, newlist of pending rules)

getsubrule([R|P],R,P).
getsubrule([R1|P],R,[R1|P1]):- getsubrule(P,R,P1).

concat([],Y,Y).
concat([X|R],Y,[X|Z]):- concat(R,Y,Z).
```

Figure 8: Top-Down Depth-First Parser

This interpreter maintains a list of pending rules which acts as a multiset of local trees that need to be used in the derivation. It tries to expand the leftmost nonterminal symbol in a derivation sequence by either invoking a type-one rule or by using a subrule of some type-two rule.[6] This same strategy has been incorporated into an SDG compiler which is discussed in section 6.3.

---

[6]The original interpreter incorporated a strategy whereby a symbol other than the leftmost nonterminal symbol could be expanded by some pending rule. This resulted in spurious ambiguities since a nonterminal symbol could in some cases be expanded either when it was the leftmost nonterminal in the derivation or just after the subrule responsible for its expansion was introduced into the pending list.

Invoking a type-one rule corresponds to associating a single local tree with the current node in the derivation tree, while invoking a type-two rule involves using the first local tree from the multiset associated with the type-two rule and adding the other local trees associated with the rule to the *pending* multiset. The third clause of the interpreter removes one of the local trees from the pending multiset via the *getsubrule* relation, and applies the subrule.

The interpreter pools all the pending subrules in a single list without preserving information about their relative order in the type-two rule from which they were obtained. This technique clearly does not incorporate the restriction $\alpha_i << \alpha_j$ from our formal definition. This is illustrated by the interpretation of the grammar introduced in (17), which will describe the language a^{2n} according to the interpretation scheme used by this parser.

(17) { s → [a], s
s → [a], s }

s → [ ]

The derivation tree for the sentence *a a* is given in Figure 9.

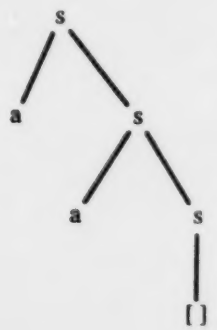

Figure 9: Derivation Tree for "a a"

As with most top-down depth-first parsers, this interpreter cannot use left recursive grammars. The << relation could be incorporated into the parser by having a more structured pending list, like that used in the left-corner parser which we shall now examine.

6.2. Left-Corner Parsing

Another approach to processing SDGs involves modifying a traditional *left-corner* parser (Rosenkrantz and Lewis, 1970) so that it maintains the pending rules associated with type-two rules instead of pooling them all together. In this section, we will discuss a Prolog left-corner SDG parser that was inspired by a left-corner Definite Clause Grammar parser described in (Pereira and Shieber, 1987).

To understand the operation of the left-corner parser, it is necessary to understand the notion of *left-corner*. For any parse tree T, a node β is a left corner of T if (i) it is the root node of T, or (ii) β is the leftmost daughter of some node α in T and α is a left corner of T. Our left-corner parser works by progressively building larger subtrees rooted by the left-corner nodes of the complete parse tree.

```
%%%  Internal Format of SDG rules.
%%%
%%%    a --> b, c.                          = sdrule(a, [b,c]).
%%%
%%%  { a --> b,c;  p --> q,r;  x --> y,z } = sdrule([[a,b,c],[p,q,r],[x,y,z]]).

lcparse(Cat, [Symbol|RestPhrase], PhraseOut, PendIn, PendOut) :-
  lc(Symbol, Cat, RestPhrase, PhraseOut, PendIn, PendOut).

lcparse(Cat, PhraseIn, PhraseOut, PendIn, PendOut) :-
  lc([], Cat, PhraseIn, PhraseOut, PendIn, PendOut).

lc(Cat, Cat, Phrase, Phrase, Pend, Pend).   % if we're at the root, it's a left-corner

lc(SubCat, SuperCat, PhraseIn, PhraseOut, PendIn, PendOut) :-
  getrule(SubCat, Cat, RestRule, Sub, PendIn, Pend1),
  link(Cat, SuperCat),
  parse_rest(RestRule, PhraseIn, Phrase, Pend1, Pend2),
  newpending(Sub, Pend2, Pend3),
  lc(Cat, SuperCat, Phrase, PhraseOut, Pend3, PendOut).

parse_rest([], Phrase, Phrase, Pend, Pend).

parse_rest([Symbol|Rest], PhraseIn, PhraseOut, PendIn, PendOut) :-
  lcparse(Symbol, PhraseIn, Phrase, PendIn, Pend),
  parse_rest(Rest, Phrase, PhraseOut, Pend, PendOut).

getrule(LC, NT, RestRule, [], Pend, Pend) :-                % type 1 rule
   sdrule(NT, [LC|RestRule]).

getrule(LC, NT, RestRule, Subrules, Pend, Pend) :-          % type 2 rule
   sdrule([[NT,LC|RestRule]|Subrules]).

getrule(LC, NT, RestRule, Subrules, PendIn, PendOut) :-     % Pending Rule
   getsubrule(PendIn, [[NT,LC|RestRule]|Subrules], PendOut).

newpending([], Pend, Pend) :- !.
newpending(Subrules, Pend, [Subrules|Pend]).
```

Figure 10: A Left Corner Parser

The Prolog left-corner SDG parser introduced in Figure 10 works by proving that a node is a left corner of a tree. The definition of the *lc* relation is used to prove that a node β is a left corner of the tree rooted by ν; β and ν are associated with the first and second arguments of *lc* respectively. The third and fourth arguments of the *lc* relation are used to represent the terminal nodes of the tree rooted by ν that are *not* contained in the subtree dominated by the left corner β; intuitively, they are input words that have not yet been analyzed by the parser. Taking the difference of these two argument will yield this list of terminal nodes. Finally, the fifth and sixth arguments of the *lc* relation are used for manipulating pending subrules that are associated with type-two rules. Shortly, we will discuss in detail the structure of the *pending list* for our left-corner parser.

The first clause of the definition of *lc* incorporates the notion that the root node of a tree is a left-corner of the tree. If the node of interest is not the root then the second clause is invoked to

prove that *SubCat* is a left-corner of the tree rooted by *SuperCat*. Ignoring a few details, this is done by first proving that there is some node *Cat* which has the node *SubCat* as its leftmost daughter; this is done by finding a rule that has *Cat* on its left hand side and *SubCat* as the leftmost symbol on its right hand side. Then, after proving that there is a subtree dominated by *Cat* (by proving that there is a parse tree dominated by each of the other symbols introduced on the right hand side of the rule for *Cat*), we need only determine that *Cat* is a left-corner of the tree rooted by *SuperCat*. The relationship between *SubCat*, *Cat* and *SuperCat* is summarized in Figure 11.

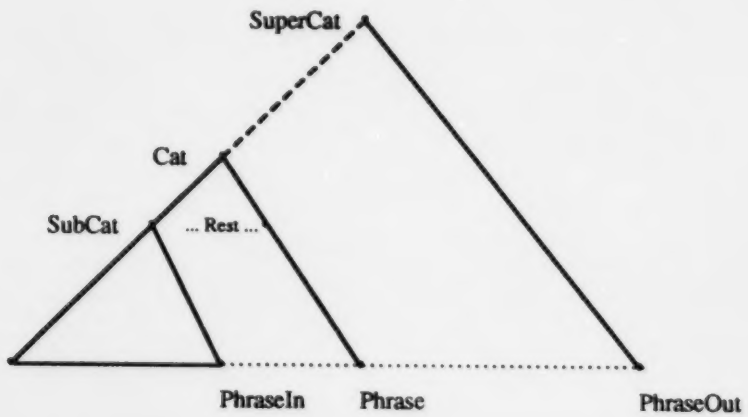

Figure 11: Left-Corner Parsing

The actual parsing relationship between a root node and the leaves of a parse tree is established with the *lcparse* predicate. The first clause of *lcparse* attempts to establish the leftmost word of the input phrase as a left-corner node of a parse tree, while the second clause corresponds to the empty word (as introduced in so-called *epsilon-production* rules like $np \rightarrow [\,]$) appearing as a left-corner node.

The *link* predicate, which appears in the definition of *lc*, accesses a *link* table that contains information about whether a symbol *Cat* can ever be a left-corner of the tree rooted by *SuperCat*. Link tables are commonly used in left-corner parsers to avoid attempting analyses that are doomed to fail. For example, if our grammar is structured so that *Cat* can never be the left-corner of *SuperCat*, then we can find this out before executing *parse_rest* and without invoking a recursive call to *lc*. An inefficient but simple program that automatically builds a link table from a grammar appears in Appendix A. For any two symbols α and β, there will be an entry *link*(β,α) in the database if given the symbol α there is some sequence of rules or subrules when applied to α will yield a derivation sequence in which β appears as the leftmost symbol. The link table is based only on the *functor* of the grammar symbols; the values of the grammar symbol *arguments* are ignored. Although this may result in the link table being "overpermissive," certain problems associated with the instantiation of variables during grammar

rule application can be avoided.

Unlike the pending list discussed in conjunction with the top-down parsers introduced earlier (which contained subrules), the pending list as used with the left-corner parser contains *lists of* subrules. Each element of the pending list is a list of subrules introduced by the invocation of a type-two rule. The advantage of this type of organization is that only the first subrule of each subrule list needs to be examined when considering which pending subrule to invoke (as reflected in the final clause of the *getrule* definition); assuming the ordering relation << from section 2, we want the first subrule of a subrule list to apply before the others from the same subrule list. Furthermore, when we invoke one subrule of the form $\alpha \rightarrow \beta_1,\beta_2,...,\beta_n$, we do not want the other subrules from this rule invocation to be used in the analysis of the subtrees associated with any of the β_i's (again in accordance with the << relation). So whenever a type-two rule is invoked or a pending subrule is selected, the other associated subrules (referenced by *Sub* in the definition of *lc*) are not added to the pending list (via a call to *newpending*) until after the call to *parse_rest*.

Due to the top-down filtering of information via the link tables, the left-corner parser is much more efficient than a purely bottom-up approach. In efficiency, it is almost identical to the top-down interpreter. It has the added advantage though of being able to process left-recursive grammars. The class of analyses allowed by the parser closely corresponds to those allowed under the tree admissibility interpretation of SDGs along with the ordering relation '<<' described earlier. To get an exact correspondence to this interpretation would require additional restrictions on the manipulation of pending lists by the parser. As it stands, if a type-two rule is invoked in a call to *lc* then there is nothing to prevent subrules of this rule invocation from being used in the recursive call to *lc*. This means that the following grammar, which is similar to the one introduced in (17), will also allow sentences of the form a^{2n} to be parsed.

(18) { s → s, [a]
s → s, [a] }

s → [ ]

Note that the left-corner parser will **not** parse any (non-empty) sentences if given grammar (17); the derivation tree from Figure 9 will not be obtained.

6.3. Compiling

One of the attractions of logic grammars is that, in theory, the same grammar can act as an analyzer or as a generator, according to whether the call specifies a known semantic representation or a known input sentence. In practice, because of operationally motivated compromises such as the use of cut, large systems do not generally achieve this goal. Still, we would like to preserve for SDGs the potential that reversibility offers to logic grammars in

general. In this section we introduce a compiler that achieves this goal.

Consider again grammar (6), which, as we said, accepts, under the tree admissibility interpretation, a number of indexed *a*'s followed by the same number of indexed *b*'s, with no concern for index order. Grammar (7) resorts to the low-level concept of tree depth to enforce the same ordering on the two sets of indices.

With grammar reversibility at our disposal, we could describe the ordered version of the grammar in higher level terms, e.g., specifying a semantic representation such as the ordered list of indices, in an extra argument as shown in (19).

(19) s(Indices) → as(Indices), bs(Indices)

{ as([ ]) → [ ]
  bs(([ ]) → [ ] }

{ as([X|L]) → [a(X)], as(L)
  bs(([X|L] → [b(X)], bs(L) }

In parsing, the set of ordered indices is obtained as a side product of string acceptance. In generation, the given ordering specified in the extra argument is enforced both upon the string of *a*'s and upon the string of *b*'s.

SDG compilers to date all follow a generalization of a technique first developed for Metamorphosis Grammars (Colmerauer, 1978), of storing the non-leading left-hand side symbols in lists, translating only the leading one into a clause head, and arranging for future use of the stored lists of pending symbols – thus keeping within Horn clause syntax. This technique, generalized by Dahl for compiling Discontinuous Grammars, resulted in a compiler called SYNAL, coded by McCord and tested within joint work on coordination. SYNAL has to deal explicitly with skips and must insert these lists of pending symbols at appropriate places. For SDG compilation, the process is much simpler due to the lack of explicit skips. The leading *subrule* introduces a Horn clause and the rest of the subrules are stored and passed around in arguments until actually applied. SDG compilers that use this technique can be made to treat both analysis and generation from the same grammar.

One such compiler, outlined in (Saint-Dizier, 1988), was developed for a subclass of SDGs that includes linear precedence restrictions and modalities. This compiler was adapted by Brown (1987) for use in generating Spanish clitics. Both this compiler and its adaptation by Brown only cover SDG rules restricted to two subrules.

```
/* Compiler to produce Prolog calls from SDG rules.
*  SDG notation (input): type-1 rules: nt --> RHS
*                        type-2 rules: [(nt1-->RHS1),...,(ntn-->RHSn)]
*/
:- op(500,xfx,\).

sdg2pl( (Nt-->RHS) ) :- !, build((Nt-->RHS),P,P,_).
sdg2pl( [SubRule|Rest] ) :-
        addvar(Rest,RestV,V),
        build(SubRule,P\V,P\RestV,_),
        dequeue(Rest).

dequeue([]).
dequeue([SubRule|Rest]) :-
        build(SubRule,P1\T,P\T,(delete(SubRule,P,P1))),
        dequeue(Rest).

build(Rule,Pb,Ph,Body2) :-
        expand_term(Rule,(Head:-Body)),
        Head =.. [Pred|Args],
        concat_var(Args,[Ph,P],NewArgs),
        Head1 =..[Pred|NewArgs],
        new_body(Pb,P,Body,Body1),
        combine(Body2,Body1,BodyFinal),
        formclause(Head1,BodyFinal,Clause),
        numbervars(Clause,0,_),
        pp(Clause).

new_body(P,P,true,true) :- !.
new_body(P0,P,('C'([A2|A3],A2,A3),S),NewS) :-
        !, new_body(P0,P,S,NewS).
new_body(P0,P,(S1,S),(NewS1,NewS)) :-
        !, S1 =..[Pred|Args],
        concat_var(Args,[P0,P1],NewArgs),
        NewS1 =..[Pred|NewArgs],
        new_body(P1,P,S,NewS).
new_body(_,_,'C'(A1,A2,A3),'C'(A1,A2,A3)) :- !.
new_body(P0,P,S,NewS) :-
        S =.. [Pred|Args],
        concat_var(Args,[P0,P],NewArgs),
        NewS =.. [Pred|NewArgs].

combine(V,B,B) :- var(V), !.
combine(A,true,A) :- !.
combine((S,S1),S2,(S,S3)) :- !, combine(S1,S2,S3).
combine(S1,S2,(S1,S2)).

formclause(Head,true,Head) :- !.
formclause(Head,Body,(Head :- Body)).

% Adds a variable at the end of a list.
addvar([],V,V).
addvar([E|R1],[E|R2],V) :- addvar(R1,R2,V).

concat_var([],L,L) :- !, nonvar(L).
concat_var([E|L1],L2,[E|L3]) :- concat_var(L1,L2,L3).

pp((A:-B)) :- !, nl, nl, writeq(A), write(':'), write('-'), nl, ppr(B), write('.'),nl.
pp(A) :- nl, nl, writeq(A), write('.'),nl.

ppr((B,C)) :- !, tab(6), writeq(B), write(','), nl, ppr(C).
ppr(B) :- tab(6), writeq(B).
% utilities to test.
grammar([ (s-->as,bs,cs,ds), [(as-->[]),(cs-->[])], [(bs-->[]),(ds-->[])],
          [(as-->[a],as),(cs-->[c],cs)], [(bs-->[b],bs),(ds-->[d],ds)]]).
grammar([ (s-->as,bs,cs), [(as-->[]),(bs-->[]),(cs-->[])] ,
          [(as-->[a],as),(bs-->[b],bs),(cs-->[c],cs)]]).

run :- grammar(G), process(G), fail.

process([]).
process([R1|Rs]) :- sdg2pl(R1), process(Rs).
```

Figure 12: SDG Compiler

The compiler shown in Figure 12, which was informally presented in (Dahl, 1989), also adapts this SYNAL technique, but admits any number of subrules in a type-two rule. It also produces more efficient code through, for instance, the use of difference-lists for representing the pending lists of rules, and the avoidance of calls induced by terminal symbols. The standard difference-list notation with "\" has been preserved for readability, but replacing it with a comma (i.e., storing a pending list and its tail in two different arguments) is better from the point of view of speed.

In producing a clause from a rule, the compiler augments each nonterminal symbol with the standard input/output string arguments, while subsuming each terminal symbol into these arguments. For instance, a rule

(20) s → [a], as, [b], bs, [c], cs

generates a clause:

(21) s([a|S1], OutputString):- as(S1,[b|S2]), bs(S2,[c|S3]), cs(S3,OutputString).

It also adds the pending-subrules input and output arguments, in difference-list notation, to the predicates springing from nonterminal symbols.

Since the string argument addition is a well-known logic grammar compiling technique (see for instance (Sterling and Shapiro, 1986)), we shall now disregard it and simply show how to go from SDG rules to Definite Clause Grammar (DCG) rules (rules with only one left-hand side symbol). In appendix B, full translations are shown.

The initial call to a grammar starts with an empty input difference-list of pending subrules, and requires its output pending list to be "terminally" empty. For s as initial symbol, we have:

(22) s(P\P, []_).

Type-one rules translate straightforwardly, with each nonterminal symbol going from one pending list to the next. For example (20) becomes:

(23) s(P,P3) --> [a], as(P,P1), [b], bs(P1,P2), [c], cs(P2,P3).

Type-two rules each generate one DCG rule per subrule. Two cases arise:

- A leading subrule originates a DCG rule which applies it and stores all other subrules in the pending list (by unifying its tail with these subrules).
- A non-initial subrule becomes a DCG rule which looks for that subrule in the pending list and creates an output pending list by deleting the subrule applied from the input pending list.

For instance, a rule like (24)

(24) { as → [a], as
bs → [b], bs
cs → [c], cs }.

yields:

```
(25) as(P\[(bs-->[b],bs),(cs-->[c],cs)|NewTail],Pending) -->  [a],
                                                              as(P\NewTail,Pending).

     bs(P\T,Pending) -->    {delete((bs-->[b],bs),P,P1)},
                            [b], bs(P1\T,Pending).

     cs(P\T,Pending) -->    {delete((cs-->[c],cs),P,P1},
                            [c], cs(P1\T,Pending).
```

The complete translation of this rule into Prolog clauses, including input and output strings, is shown in appendix B.

The compiler parameterizes these three cases (type-one rule, leading subrule, non-leading subrule) into the central predicate

(26) *build(Rule,Pb,Ph,B)*

where *Pb* is the input pending list of subrules for the body of the new clause, *Ph* is the input pending list of subrules for its head, and *B* contains the appropriate call to *delete* when *Rule* is a non-leading subrule.

The predicate *build* starts by standardly transforming the rule or subrule *Rule* into a Horn clause (through the system predicate *expand_term*), and completes the clause in appropriate ways for each type of rule, as exemplified above. The topmost predicate, *sdg/1*, issues one call to *build* for a type-one rule, and (using the auxiliary predicate *dequeue*), calls *build n* times for a type-two rule with *n* subrules.

7. Concluding Remarks

We have shown how the introduction of static discontinuity into logic grammars can be useful both in linguistic and non-linguistic applications, and how it allows for transparent constraints on derivation trees, thus lending itself to straightforward automations of linguistic constraint descriptions. We have also presented SDG processors with some attractive features.

It is important to stress that SDGs are no more than convenient *tools* to make executable linguistic descriptions easier, and that an SDG implementation of a complete "correct" version of

any one theory (if such a thing exists) still remains to be done. Nevertheless, SDGs have already proved useful for describing some important principles of GB theory (Brown et.al., 1986, Brown, 1987, Dahl, 1988c, Dahl, 1989).

As we said in the introduction, our techniques not only apply to static discontinuity *grammars*, but to their logic programming version, *discontinuous logic*. This transfer of the SDG idea into logic programming proper consists of grouping several definite clause-like subrules to be applied on matching discontinuous leaves of a proof tree, all sharing the same substitution (Dahl et.al., 1986). In other words, *n* Horn-clause like rules with heads h_i and bodies b_i can be grouped as indicated by the curly brackets in (27) to provide context to each other through shared substitutions.

(27) { h_1 :- b_1
h_2 :- b_2
.
.
.
h_n :- b_n }

These groups of clauses now apply on proof states rather than derivation trees. For instance, our permutation problem of section 5.1 can be expressed in Discontinuous Logic as follows (we simply add input and output string arguments):

```
(28)  s(S1,S) :- movable(a,S1,S2), movable(b,S2,S3), movable(c,S3,S4), s(S4,S).
      s(S,S).

      movable(X,[X|S],S).

    { movable(X1,[X2|S],S) :- diff(X1,X2).
      movable(X2,[X1|S],S) }
```

The idea of Discontinuous Logic is followed up in DISLOG (Saint-Dizier, 1987). Distributed Logic (Monteiro, 1981) can be considered an antecedent of Discontinuous Logic, within the area of concurrent programming and with a fixed parallel strategy for applying the subrules.

Acknowledgements

We would like to thank all the people who, as either members of or visitors to V. Dahl's research group, have contributed in one way or another to the development and testing of various applications and processing schemes for Static Discontinuity Grammars: Michel Boyer, Charles Brown, Sharon Hamilton, Diane Massam, Pierre Massicotte, Brenda Orser, T. Pattabhiraman and Patrick Saint-Dizier. One of these applications was supported by an SUR research contract from IBM Canada.

We would also like to thank Dan Fass and Carl Vogel for useful comments on this article's first draft, and Pierre Massicotte, for his help in testing and debugging the compiler.

References

Aho, A.V. and Ullman, J.D. *The Theory of Parsing, Translation, and Compiling, Volume 1: Parsing*. Prentice-Hall, 1972.

Brown, C. *Generating Spanish Clitics using Static Discontinuity Grammar*. PhD thesis, Simon Fraser University, 1987.

Brown, C., Dahl, C., Massam, D., Massicotte, P., Pattabhiraman, T. *Tailoring Government and Binding Theory for Use in Natural Language Translation*. Technical Report LCCR 86-4, Simon Fraser University, 1986.

Chomsky, N. *Lectures on Government and Binding, the Pisa Lectures, 2nd (revised) Edition*. Foris Publications, Holland, 1982.

Colmerauer, A. Metamorphosis Grammars. In L. Bolc (Ed.), *Lecture Notes in Computer Science*, Springer-Verlag, 1978.

Dahl, V. *More on Gapping Grammars*. Proceedings International Conference on V Generation Computer Systems, Tokyo, 1984.

Dahl, V. Gramaticas discontinuas: una herramienta computacional con aplicaciones en la teoria de Reccion y Ligamiento. *Revista Argentina de Lingüistica*, 1986, *2*(2), 375-392.

Dahl, V. *Discontinuous Grammars*. Technical Report CSS/LCCR 88-26, Simon Fraser University, 1988.

Dahl, V. *Representing Linguistic Knowledge through Logic Programming*. Fifth International Conference/Symposium on Logic Programming, Seattle, August, 1988.

Dahl,V. *Static Discontinuity Grammars for Government and Binding Theory*. Proc. Workshop 'Informatique and langue naturelle', Université de Nantes, October, 1988.

Dahl, V. Compiling Static Discontinuity Grammars. Extended abstract presented at the Informal Workshop on Prolog as the Implementation Language for Natural Language, Sweden, 1989.

Dahl, V. and Abramson, H. *On Gapping Grammars*, pages 77-88. Proceedings, Second International Logic Programming Conference, Uppsala, Sweden, 1984.

Dahl, V. and Massicotte, P. *Meta-Programming for Discontinuous Grammars*. Proceedings of the Meta-Programming for Logic Programming Workshop, University of Bristol, 1988.

Dahl, V. and McCord, M. Treating coordination in logic grammars. *American Journal of Computational Linguistics*, 1983, *9*, 69-91.

Dahl, V. and Saint-Dizier, P. *Constrained Discontinuous Grammars - A linguistically motivated tool for processing language*. Technical Report LCCR 86-11, Simon Fraser University, 1986.

Dahl, V., Brown, C. and Hamilton, S. *Static Discontinuity Grammars and Logic Programming*. Technical Report LCCR 86-17, Simon Fraser University, 1986.

Gazdar, G., Klein, E., Pullum, G. and Sag, I. *Generalized Phrase Structure Grammar*. Basil

Blackwell, London, 1985.

Greibach, S. and Hopcroft, J. Scattered Context Grammars. *Journal of Computer and System Sciences*, 1969, (3), 233-247.

Knuth, D.E. *The Art of Computer Programming, Volume 2.* Addison-Wesley, 1969.

Monteiro, L. *Distributed Logic: A Logical System for Specifying Concurrency.* Technical Report 5/81, Centro de Informatica, Universidade Nova de Lisboa, 1981.

Pereira, F.C.N. Extraposition Grammars. *American Journal for Computational Linguistics*, 1981, *7*, 243-256.

Pereira, F.C.N. and Shieber, S. *Prolog and Natural Language Analysis.* Center for the Study of Language and Information, Stanford, 1987.

Pereira, F.C.N. and Warren, D.H.D. Definite Clause Grammars for Language Analysis - A survey of the formalism and a Comparison with Transition Networks. *Artificial Intelligence*, 1980, *13*, 231-278.

Popowich, F. Unrestricted Gapping Grammars. *Computational Intelligence*, 1986, *2*(1), 28-53.

Rosenkrantz, D.J. and Lewis II, P.M. *Deterministic Left Corner Parser*, pages 139-152. IEEE Conference Record of the 11th Annual Symposium on Switching and Automata Theory, 1970.

Saint-Dizier, P. *Dislog: Programming in Logic with Discontinuities.* Technical Report LCCR 87-13, Simon Fraser University, 1987.

Saint-Dizier, P. Contextual Discontinuous Grammars. In Dahl and St-Dizier (Ed.), *Natural Language Understanding and Logic Programming II*, North-Holland, 1988.

Sterling, L., and Shapiro, E. *The Art of Prolog.* The MIT Press, 1986.

N.B. In some early reports, SDGs were either unnamed as in (Dahl, 1986), or were inappropriately called Constrained Discontinuous Grammars as in (Brown, 1987, Dahl et.al., 1986). To avoid confusion, the misnaming in these reports was corrected by the authors once a suitable name was coined, since Constrained Discontinuous Grammars is in fact a different, existing formalism (Dahl and Saint-Dizier, 1986), which simply augments general Discontinuous Grammars with constraints, and does not possess the static discontinuity feature. Its constraints have served as ancestors of SDG constraints, but are in fact quite different, since, among other things, the latter exploit the feature of unmovable skips for achieving power that cannot be attained in Constrained Discontinuous Grammars.

A. Automatic Generation of Link Tables

```
makelink  :- link(_,_), !.                                  % quit if table exists
makelink  :- assertz(link(X,X)), fail.                      % final axiom: it's reflexive
makelink  :- sdrule(Lhs, [Sym|_]),  islink(Sym,Lhs), fail. % For type one rules...
makelink  :- sdrule(List), makelink(List), fail.            % For type two rules...
makelink  :- transc.                                        % Perform transitive closure of link

makelink([]) :- !.
makelink([[Lhs,Sym|_]|Rest]) :- islink(Sym,Lhs), fail.      % Add a link for each subrule
makelink([_|Rest]) :- makelink(Rest).

islink(A,B)  :-
   A \== [],
   free(A, NewA),                  % replace all arguments with variables
   free(B, NewB),
   \+ link(NewA, NewB),            % does such a link already exist?
   assertz(link(NewA,NewB)), !.    % if not, add it to the database

transc :-                          % perform an inefficient transitive closure
   link(A, B),
   link(B, C),
   \+ link(A, C),                  % make sure such a link does not already exist
   \+ new(link(A,C)),              % make sure such a link is not already a potential new link
   assertz(new(link(A,C))),        % we have a potential new link
   fail.

transc :- \+ new(_), !.            % if the above did not produce any new links, we are done

transc :- retract(new(link(A,C))), assertz(link(A,C)), fail. % otherwise assert all the new links
transc :- transc.                                            % ... and try again

varlist([],[]).
varlist([_|Rest],[_|Result]) :- varlist(Rest,Result).

free(Term,NewTerm) :-  Term =.. [F|Args],  varlist(Args, NewArgs), NewTerm =.. [F|NewArgs], !.
```

Errata for CSS/LCCR TR 89-10

The definition of the predicate **delete**, given on pages 26 and 27, should be replaced by:

```
delete(E,L,L1):- nonvar(L), del(E,L,L1).

del(E,[E|L],L):- !.
del(E,[E1|L],[E1|L1]):- delete(E,L,L1).
```

B. Examples of Input Grammar and Output Code for the Compiler

Sample Inputs:

Input Grammar 1: $L(G) = \{a^n b^n c^n \mid n \geq 0\}$

```
[   (s --> as,bs,cs),
    [(as --> []), (bs --> []), (cs --> [])] ,
    [(as --> [a],as), (bs --> [b],bs), (cs --> [c],cs)]  ]
```

Input Grammar 2: $L(G) = \{a^n b^m c^n d^m \mid n,m \geq 0\}$

```
[   (s --> as,bs,cs,ds),
    [(as --> []), (cs --> [])],
    [(bs --> []), (ds --> [])],
    [(as --> [a],as), (cs --> [c],cs)],
    [(bs --> [b],bs), (ds --> [d],ds)]  ]
```

Sample Outputs:

Output Prolog Program for Grammar 1:

```
s(A,B,C,D) :-
        as(A,E,C,F),
        bs(E,G,F,H),
        cs(G,B,H,D).

as(A,A,B\[(bs-->[]),(cs-->[])|C],B\C).
as([a|F],B,C\[(bs-->[b]','bs),(cs-->[c]','cs)|D],E) :-
        as(F,B,C\D,E).

bs(A,A,B\C,D\C) :- delete((bs-->[]),B,D).
bs([b|G],B,C\D,E) :-
        delete((bs-->[b]','bs),C,F),
        bs(G,B,F\D,E).

cs(A,A,B\C,D\C) :- delete((cs-->[]),B,D).
cs([c|G],B,C\D,E) :-
        delete((cs-->[c]','cs),C,F),
        cs(G,B,F\D,E).

delete(E,[E|L],L) :- !.
delete(E,[E1|L1],[E1|L2]) :- delete(E,L1,L2).
```

Output Prolog Program for Grammar 2:

```
s(A,B,C,D) :-
        as(A,E,C,F),
        bs(E,G,F,H),
        cs(G,I,H,J),
        ds(I,B,J,D).

as(A,A,B\[(cs-->[])|C],B\C).
as([a|F],B,C\[(cs-->[c]',' cs)|D],E) :-
        as(F,B,C\D,E).

cs(A,A,B\C,D\C) :- delete((cs-->[]),B,D).
cs([c|G],B,C\D,E) :-
        delete((cs-->[c]',' cs),C,F),
        cs(G,B,F\D,E).

bs(A,A,B\[(ds-->[])|C],B\C).
bs([b|F],B,C\[(ds-->[d]',' ds)|D],E) :-
        bs(F,B,C\D,E).

ds(A,A,B\C,D\C) :- delete((ds-->[]),B,D).
ds([d|G],B,C\D,E) :-
        delete((ds-->[d]',' ds),C,F),
        ds(G,B,F\D,E).

delete(E,[E|L],L) :- !.
delete(E,[E1|L1],[E1|L2]) :- delete(E,L1,L2).
```

Call:

```
s(InputString, OutputString, P\P, []\_).
```

Centre for Systems Science

• Simon Fraser University established the **Centre for Systems Science** as a provincial centre for advanced study. The objective of CSS is twofold: coordinate and combine advanced systems research at SFU, and encourage application of research results within industry in order to further the technological development of British Columbia.

•CSS has assembled three research groups to streamline its efforts. The orientation of these groups is as follows: intelligent systems, microelectronics, and computer and communications systems. Because of shared research interests, this latter group merges with LCCR (**Laboratory for Computer and Communications Research**) for the publication of technical reports.

• CSS's three groups involve over 80 skilled faculty researchers principally from SFU's Schools of Engineering Science and Computing Science, but also draw substantially on Mathematics, Physics, Geography, Philosophy, Linguistics, Psychology, Communication, and Kinesiology.